Joseph Farmer, Connor Finley
CIS 4930 (Introduction to A.I.)
Project 1

Analysis of the A* Search Algorithm Using Two Different Heuristics

I. Introduction

The purpose of the project was to design and implement a program capable of finding an optimal path within a graph using the A* search algorithm. The program's input includes files that contain lists of node names, their locations, and their connections to each other. Additionally, the user can specify whether to display the pathway results in a step-by-step manner or an answer-only manner.

The A* search algorithm is largely based on an estimated *total* cost function, $f(n)$, according to the following equation:

$$f(n) = g(n) + h(n) \qquad \text{(Eq. 1)}$$

Here, $g(n)$ is the cost of searching from the start node to the current node, and $h(n)$ (the heuristic) is the *estimated* cost from the next, potential node to the target/end node. Two different heuristics were used in the implementation of this algorithm. The first, "straight-line distance," uses the cartesian distance formula as the estimation, $h(n)$. The second heuristic, "fewest links," uses a constant value (0 in this case) to estimate the remaining distance.

Beginning with the current node, the search algorithm calculates $f(n)$ for each neighboring node and chooses the one with the lowest value as the next step in the pathway. Note that, in most cases, both heuristics mentioned above serve as *under*estimates of the "true" remaining cost, $C(n)$. They are only equal to the true cost when the current node is directly adjacent to the target node. In other words, $h(n)$ only tells the truth on the last step of the search, and it gives an underestimate everywhere else. This is significant when considering the mechanism of the search algorithm, itself. In equation 1, the heuristic $h(n)$, can be either less than, equal to, or greater than the true remaining cost, $C(n)$. If $h(n) \leq C(n)$, the algorithm will be more investigative in its search, and is therefore guaranteed to find the optimal pathway. If $h(n) = C(n)$, the algorithm always guesses correctly and never needs to discover other possible pathways. Finally, if $h(n) > C(n)$, the algorithm will not spend much time discovering multiple pathways. As a result, it will return a pathway quickly, but will rarely be the optimal one. The heuristic demands a tradeoff between the speed of finding a solution and the correctness or quality of that solution. A* search requires that the heuristic always be less than the true cost. This guarantees that the pathway found will be optimal.

In this experiment, the Python language was used to implement the algorithm described above. The critical data structure used is a dictionary containing the open set of found nodes which have not yet been analyzed. As the search progresses, new nodes are added to the open set or eliminated from it depending on their estimated cost functions. The loop is terminated only when this open set is empty, which occurs when a path is not found. Another dictionary is used to store the optimal pathway during the process.

An example set of input files were provided for debugging. They were used in the testing and performance section of this experiment. Their data included a list of 29 node names, their locations, and the names of other specific nodes to which they are connected. The graph below provides a visual representation of this data.

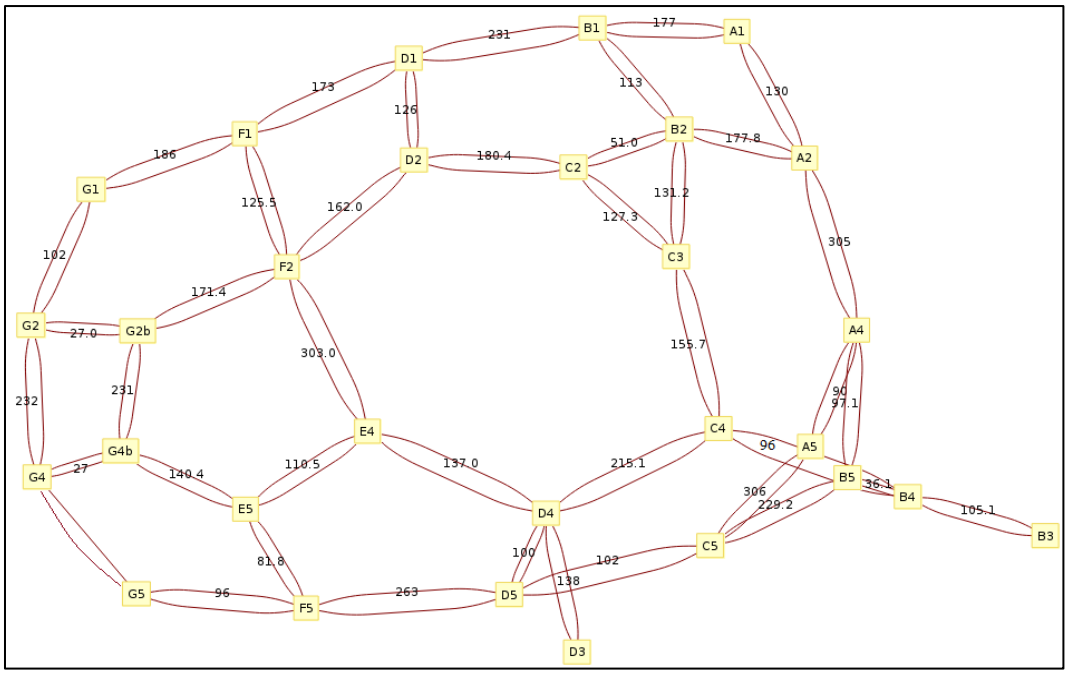

Figure 1. Visual representation of the graph data used in this experiment. Note that all edges are bidirectional. That is, there are no one-way paths initially shown.

II. Function Descriptions and Program Design

The design of this program was simple. One file, run.py, was used to capture the necessary data points from the user: the file paths to the connections and locations text files, the excluded cities, the heuristic, and how the results should be displayed. This information was passed off to search.py, which held all the logic for the A* algorithm as well as the heuristics. Such functions included a Euclidean distance calculator between two cartesian coordinates, and a method to backtrack from the target node back to the starting node, given the two were connected.

Because the typing in of all this information into the console became laborious when testing, an automated route was created in tests.py. The code in search.py was designed to be functional such that it could be decoupled from the rest of the code in run.py and could exist and run in an isolated state. Because of this, we could input dummy data into tests.py, then pass the data off to search.py. This required significantly less typing in the end. Instead of completing each user prompt for the same test case over and over, we could just change a couple of lines of code in tests.py, then run it to get the same results. Simply put, we could bypass run.py if need be.

III. Method and Results

The performance of the program was tested using multiple scenarios of user input. The resulting pathways and outputs (experimental results) were then checked against our own calculations (actual pathways). Three pairs of start and end points were tested. For each of these pairs, both heuristics were tested four times each. Each heuristic was tested with and without an excluded city, as well as with the step-by-step method and the answer-only method.

The first test consisted of a search from node D4 to G5 (See figure 2). The value of $f(n)$ was calculated for each of D4's neighbors (E4, D5, and C4) using the straight-line distance heuristic. Table 1 shows the results and the fact that node E4 gives the lowest value. After this, the same process was carried out for node E4's neighbors, and so on until arriving at the target node. This gives a total distance of 425.28 and a pathway consisting of D4->E4->E5->F5->G5 (see figures 4 – 7). Figure 8 shows an example of the output given by the program. We see that the output shown agrees with our own calculations for the algorithm.

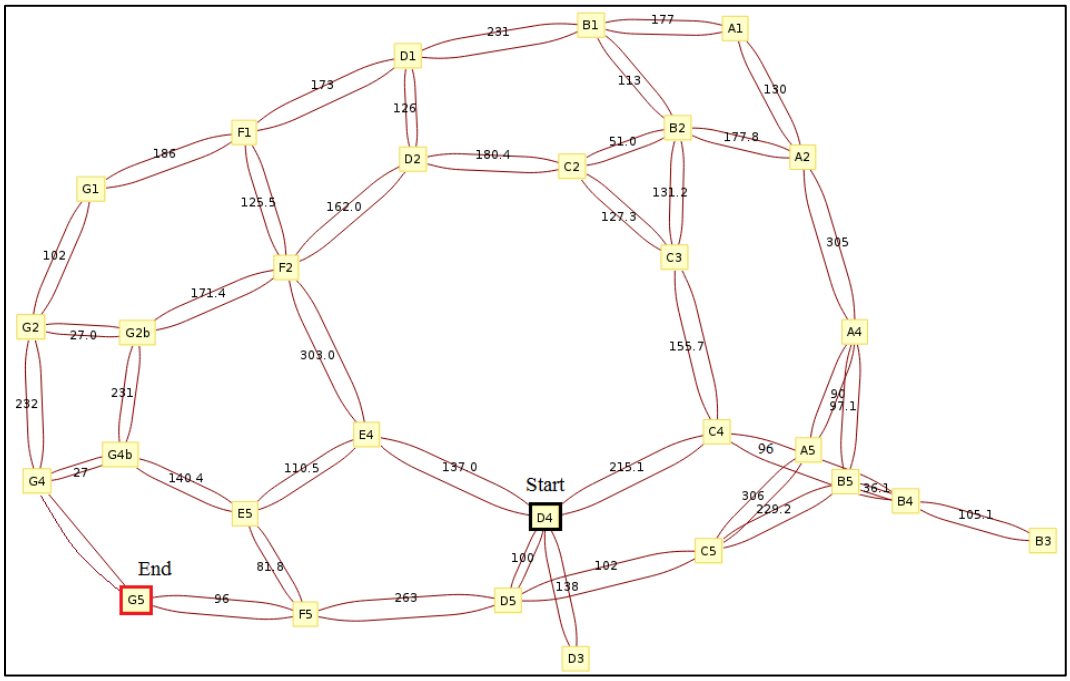

Figure 2. Graph for the first testing scenario (D4 -> G5). Start and end nodes are outlined.

	E4	D5	D3	C4
g(n)	137	100	138	215.1
h(n)	242.7	359	430.7	586.6
f(n)	379.7	459	568.7	801.7

Table 1. First iteration of search shows cost values for all four of D4's neighbors. Note that E4 has the lowest value and should be the optimal node chosen based on the straight-line distance heuristic.

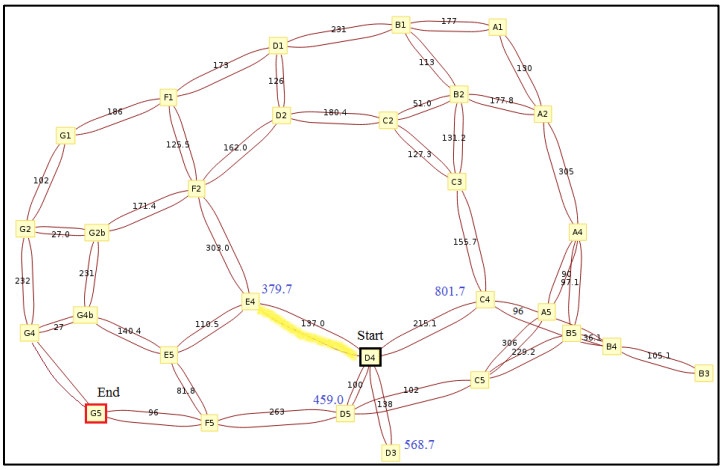

Figure 4. Step 1 of search D4->G5 using straight-line distance heuristic. $f(n)$ scores are shown in blue.

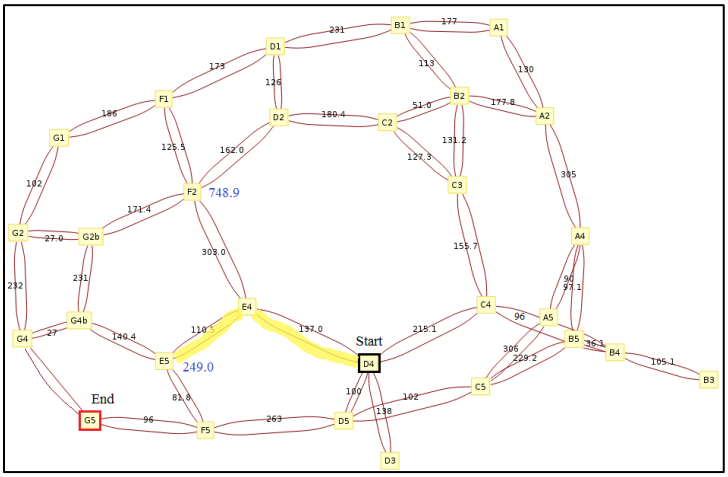

Figure 5. Step 2 of search D4->G5 using straight-line distance heuristic $f(n)$ scores are shown in blue.

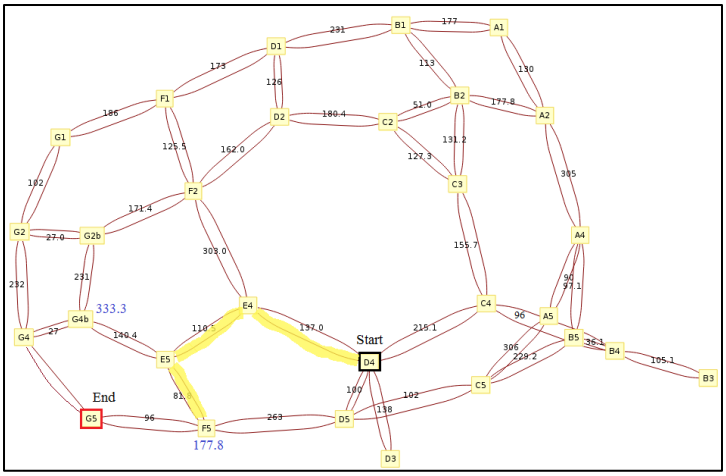

Figure 6. Step 3 of search D4->G5 using straight-line distance heuristic
$f(n)$ scores are shown in blue.

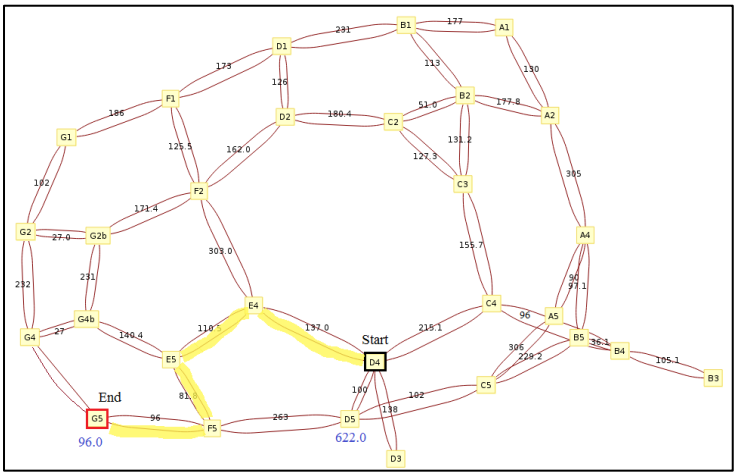

Figure 7. Last step of search D4->G5 using straight-line distance heuristic
$f(n)$ scores are shown in blue.

```
Heuristic: Straight line distance
Starting city: D4
Target city: G5
Excluding:

Final path: D4 -> E4 -> E5 -> F5 -> G5
Distance traveled: 425.28

In [50]:
```

Figure 8. Example output for the program. No excluded cities were specified. Input parameters included the straight-line distance heuristic, and the answer-only output method.

Switching to the step-by-step output method gives the same results and confirms the algorithm's functionality. Figure 4 shows example output.

```
Heuristic: Straight line distance
Starting city: D4
Target city: G5
Excluding:


Press ENTER for next step

Current path: D4
Distance traveled: 0.00
Best move is from D4 to E4

Press ENTER for next step

Current path: D4 -> E4
Distance traveled: 137.01
Best move is from E4 to E5

Press ENTER for next step

Current path: D4 -> E4 -> E5
Distance traveled: 247.49
Best move is from E5 to F5

Press ENTER for next step

Current path: D4 -> E4 -> E5 -> F5
Distance traveled: 329.28
Best move is from F5 to G5

Press ENTER for next step

Final path: D4 -> E4 -> E5 -> F5 -> G5
Distance traveled: 425.28

In [51]:
```

Figure 4. Example output for the program. Input parameters included no excluded cities, the straight-line distance heuristic, and the step-by-step output method.

Looking again at figure 2, it is obvious that using the "fewest links" heuristic should give us a different pathway. The fewest links algorithm sets the h value to 0. This then puts all the weight (f value) of a choice on the cost to move from the start to a particular node (g value). Since the f value is minimized, paths which move less from the start are prioritized.. Specifically, $f(n)$ is set to $g(n) + 1$ for all neighbors encountered. This means that generally, all nodes will be expanded in a manner similar to the BFS algorithm. In fact, the BFS algorithm is just the A* search algorithm with $h(n) = constant$ and all edges having the same value. This is evidenced by the fact that setting $h(n) = 1$ and $h(n) = 0$ gives the exact same output, which can be seen in figure 10. Although the output is the same, setting $h(n) = 0$ keeps the heuristic admissible because it never overestimates the distance to target. Setting $h(n) = 1$ on a potential choice/neighbor becomes a problem when that choice is the target node. At that point, the h value should be 0 and no more. By setting the h value consistently to 0, the heuristic is never overestimating the actual distance.

h(n) = 1	h(n) = 0
Current path: D4 -> D3	Current path: D4 -> D3
Current path: D4 -> C4	Current path: D4 -> C4
Current path: D4 -> E4	Current path: D4 -> E4
Current path: D4 -> D5	Current path: D4 -> D5
Current path: D4 -> C4 -> B4	Current path: D4 -> C4 -> B4
Current path: D4 -> C4 -> C3	Current path: D4 -> C4 -> C3
Current path: D4 -> E4 -> E5	Current path: D4 -> E4 -> E5
Current path: D4 -> E4 -> F2	Current path: D4 -> E4 -> F2
Current path: D4 -> D5 -> C5	Current path: D4 -> D5 -> C5
Current path: D4 -> D5 -> F5	Current path: D4 -> D5 -> F5
Current path: D4 -> C4 -> B4 -> B3	Current path: D4 -> C4 -> B4 -> B3
Current path: D4 -> C4 -> B4 -> B5	Current path: D4 -> C4 -> B4 -> B5
Current path: D4 -> C4 -> C3 -> C2	Current path: D4 -> C4 -> C3 -> C2
Current path: D4 -> C4 -> C3 -> B2	Current path: D4 -> C4 -> C3 -> B2
Current path: D4 -> E4 -> E5 -> G4B	Current path: D4 -> E4 -> E5 -> G4B
Current path: D4 -> E4 -> F2 -> F1	Current path: D4 -> E4 -> F2 -> F1
Current path: D4 -> E4 -> F2 -> D2	Current path: D4 -> E4 -> F2 -> D2
Current path: D4 -> E4 -> F2 -> G2B	Current path: D4 -> E4 -> F2 -> G2B
Current path: D4 -> D5 -> C5 -> A5	Current path: D4 -> D5 -> C5 -> A5
Final path: D4 -> D5 -> F5 -> G5	Final path: D4 -> D5 -> F5 -> G5

Figure 10. Simplified outputs for scenario D4 -> G5 with the fewest-links heuristic set to 1 and to 0.

Figure 11 shows the final pathway given by this heuristic. We see the path highlighted is indeed the one that passes through the fewest number of links. The experiment was repeated with node D5 excluded, and the resulting path was predictably the same as that for the straight-line distance heuristic in figure 7. An overview of the results for test scenario 1 is provided in table 2.

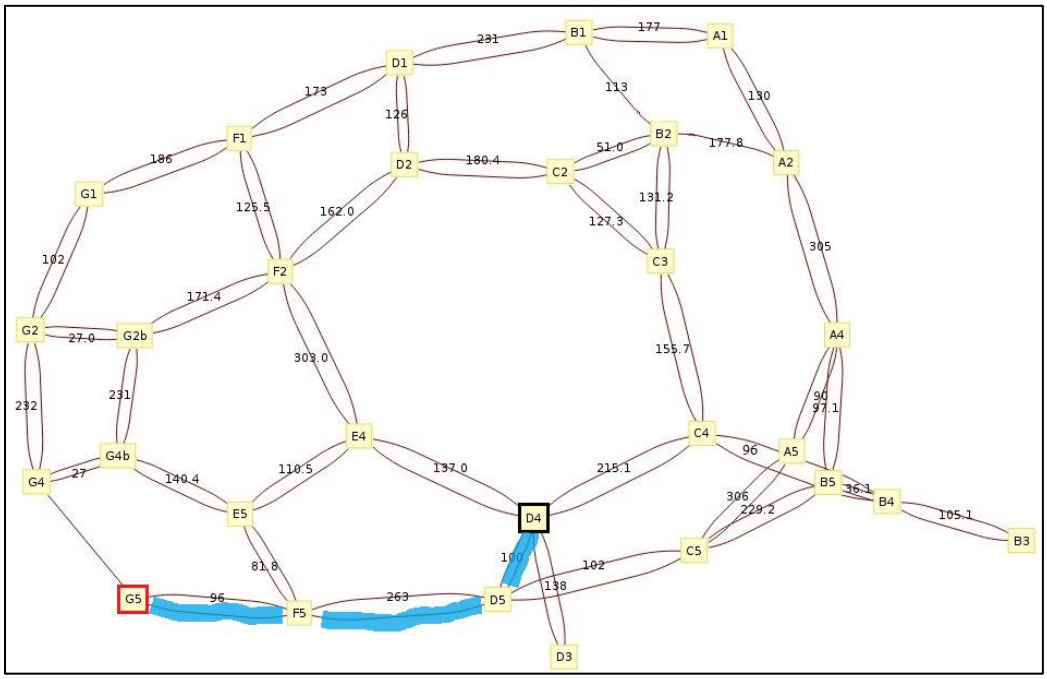

Figure 11. Final pathway for D4 -> G5 using the fewest-links heuristic.

Heuristic	Display	Start	End	Excluded	Final Measurement	Actual Measurement
straight-line	final path	D4	G5	-	425.28	425.28
straight-line	final path	D4	G5	E4	459.0	459.0
straight-line	step-by-step	D4	G5	-	425.28	425.28
straight-line	step-by-step	D4	G5	E4	459.0	459.0
fewest links	final path	D4	G5	-	3	3
fewest links	final path	D4	G5	D5	4	4
fewest links	step-by-step	D4	G5	-	3	3
fewest links	step-by-step	D4	G5	D5	4	4

Table 2. Overall results for test scenario 1 (D4 -> G5).

In the second testing scenario we analyzed the output for searching from node B1 to D4 in a manner similar to scenario 1. The results were again in agreement with our own calculations. Figures 12 and 13 show the final path and program output, respectively, for the straight-line heuristic with node B2 excluded. An overview of the results for the entire scenario are given in table 3.

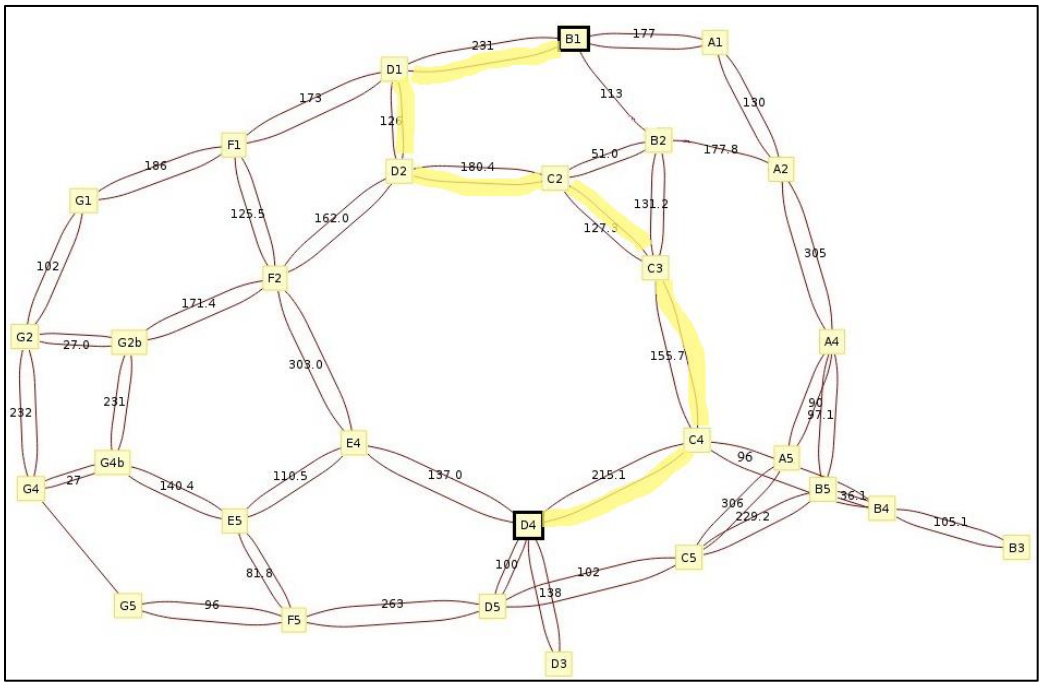

Figure 12. Resulting pathway for B1->D4 using the straight-line distance heuristic and node B2 excluded.

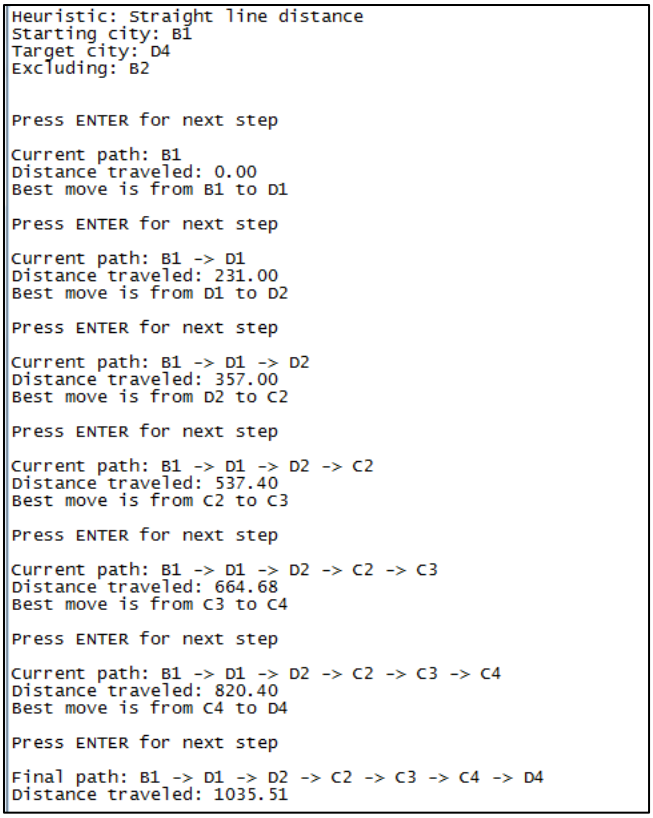

Figure 13. Output for B1->D4 using the straight-line distance heuristic and node B2 excluded.

Heuristic	Display	Start	End	Excluded	Final Measurement	Actual Measurement
straight-line	final path	B1	D4	-	615.00	615.0
straight-line	final path	B1	D4	B2	1035.5	1035.5
straight-line	step-by-step	B1	D4	-	615	615.0
straight-line	step-by-step	B1	D4	B2	1035.5	1035.5
fewest links	final path	B1	D4	-	4	4
fewest links	final path	B1	D4	B2	5	5
fewest links	step-by-step	B1	D4	-	4	4
fewest links	step-by-step	B1	D4	B2	5	5

Table 3. Overall results for test scenario 2 (B1 -> D4).

The final scenario tested was from node B2 to A2. Excluded cities were not specified in this test. Instead, we modified the input files to include one-way edges. Figure 14 shows that node B2 now has two adjacent edges which are unidirectional and pointing towards itself. The program was successfully able to navigate around the one-way edge and give an optimal path to the target node. Figure 15 gives example output using the straight-line distance heuristic, and Table 4 provides an overview of the results for this test scenario.

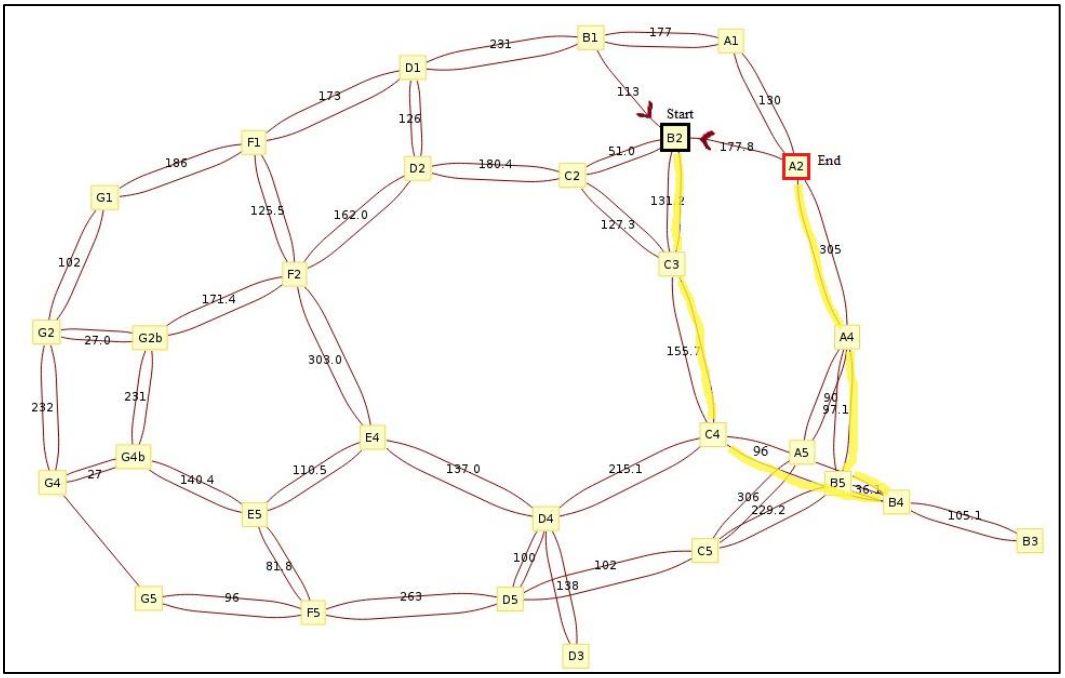

Figure 14. Graph showing the third test scenario (B2 -> A2). Note that (A2 -> B2) and (B1 -> B2) are one-way edges in the direction of node B2.

```
Heuristic: Straight line distance
Starting city: B2
Target city: A2
Excluding:

Press ENTER for next step

Current path: B2
Distance traveled: 0.00
Best move is from B2 to C2

Press ENTER for next step

Current path: B2 -> C2
Distance traveled: 51.01
Best move is from B2 to C3

Press ENTER for next step

Current path: B2 -> C3
Distance traveled: 131.22
Best move is from C3 to C4

Press ENTER for next step

Current path: B2 -> C3 -> C4
Distance traveled: 286.94
Best move is from C4 to B4

Press ENTER for next step

Current path: B2 -> C3 -> C4 -> B4
Distance traveled: 382.94
Best move is from B4 to B3

Press ENTER for next step

Current path: B2 -> C3 -> C4 -> B4 -> B3
Distance traveled: 488.02
Best move is from C2 to D2

Press ENTER for next step

Current path: B2 -> C2 -> D2
Distance traveled: 231.41
Best move is from B4 to B5

Press ENTER for next step

Current path: B2 -> C3 -> C4 -> B4 -> B5
Distance traveled: 419.00
Best move is from B5 to A4

Press ENTER for next step

Current path: B2 -> C3 -> C4 -> B4 -> B5 -> A4
Distance traveled: 516.08
Best move is from A4 to A2

Press ENTER for next step

Final path: B2 -> C3 -> C4 -> B4 -> B5 -> A4 -> A2
```

Figure 15. Output for B2->A2 using the straight-line distance heuristic and two, one-way edges to navigate around.

Heuristic	Display	Start	End	Final Measurement	Actual Measurement
straight-line	final path	B2	A2	821.1	821.1
straight-line	step-by-step	B2	A2	821.1	821.1
fewest links	final path	B2	A2	6	6
fewest links	step-by-step	B2	A2	6.0	6.0

Table 4. Overall results for test scenario 3 (B2 -> A2). In all four situations, the input file was modified to include some one-way edges for the program to navigate.

IV. Performance

The results generated by the algorithm were expedient, given the implementation was correct. At first, an incorrect data structure was used for the final path. This only affected the fewest links heuristic and is detailed in section V below. This led to an incorrect and longer path. Because of this length, the algorithm had to explore mode nodes and process more data, which resulted in a noticeable slowdown for the fewest links algorithm. That being said, the slowdown was still in the range of tens or hundreds of milliseconds. Given a larger graph, however, the incorrect implementation's shortcoming would be much more noticeable.

Because the fewest links heuristic works in a breadth-first search manner, the user does have to wait longer if going step-by-step since the nodes explored work in a spiral/level fashion: first exploring *all* the nodes one link away, then *all* the nodes two links away, and so on… Because of this, the user has to step through each level of the spiral before reaching the target. This could be helped by a look-ahead helper, which could prematurely check if a potential choice/neighbor was, in-fact, the target. If so, then a negative f value could be set and guarantee the target would be next to be picked as the current for when the loop later checks if the path has reached an end.

V. Project Experience and Task management

Connor designed and implemented the program's outline and functionality, particularly in the earlier versions of the project. He also wrote the readme file. Joseph outlined and wrote the first draft of the report. The remaining tasks involved testing and troubleshooting the algorithms with different user input combinations. Both members were equally involved in this process.

In the weeks prior to beginning the actual project, we kept moderate communication and studied the A* search algorithm independently. One notable setback occurred during this time when it was discovered that we were using two different programming languages (C++ and Python) to implement the project. We quickly realized the misunderstanding and ultimately agreed to use Python.

The most challenging part of the assignment, arguably, was trying to understand exactly which heuristic to use for the fewest links option. Searching for the answer to this question lead to a better understanding of how A*, Dijkstra's, and BFS algorithms are all related. We also learned what a large part the heuristic plays in controlling the quality and speed of a search result.

One additional hang-up that we'd like to mention was the issue of the final path data structure. Originally, we were using a serial list of nodes, appended one after the other, to make a final path. This data structure became a problem when low f-value nodes were to be chosen, but the path was already populated with nodes that would not make sense as predecessors, and would lead to an impossible path. By switching to a linked list, where a node's parent or previous move was tied in, the data structure allowed for branching to find the target. When the target was found, all that was needed was a backwards traversal up the particular branch that ended with the

starting node. All other branches were ignored. In this way, the shortest path to the target was kept in mind, and made the organization of the algorithm streamlined and accommodating.